



307307

# Introduction to Transformers and Large Language Models

# Introduction to Transformers



# Transformers – Architecture and Principles

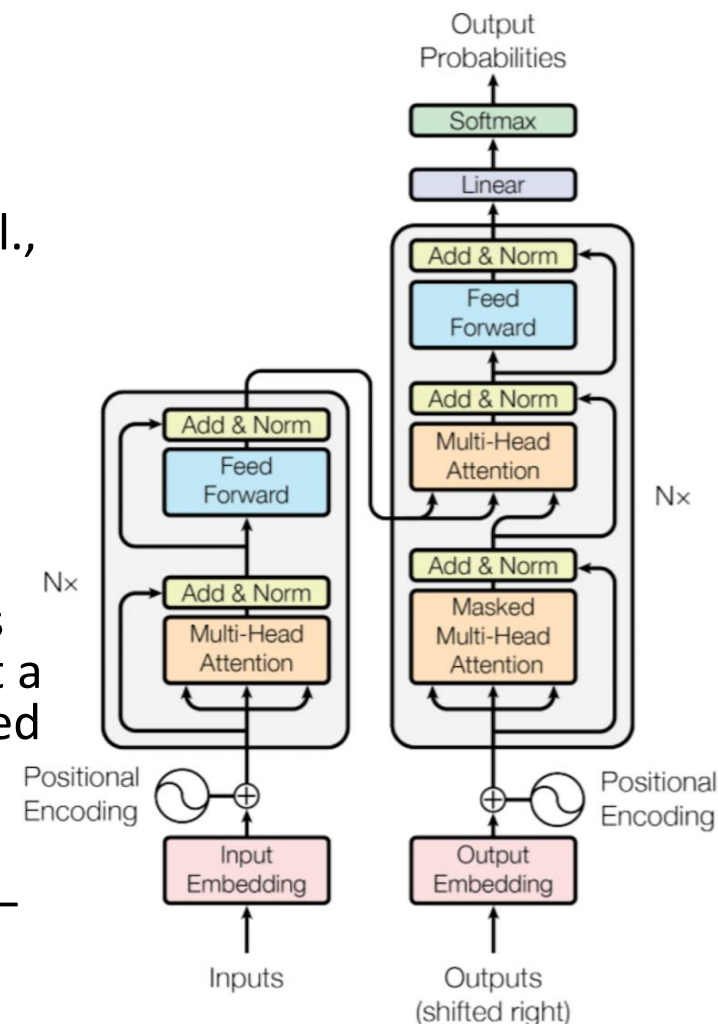
## What is a Transformer?

- A deep learning model based entirely on **self-attention**, with no recurrence or convolutions
- Introduced in the paper “*Attention Is All You Need*” (Vaswani et al., 2017)

## The transformer model consists of two main parts:

1. **Encoder:** The encoder processes the input sequence and generates a continuous representation of it. This representation captures the contextual information of the input tokens.
2. **Decoder:** The decoder takes the encoder's output and generates the final output sequence. It does this by predicting one token at a time, using the encoded representations and previously generated tokens.

Both the encoder and decoder are composed of multiple identical layers—typically six layers in the original transformer architecture—allowing for deep learning and complex feature extraction.



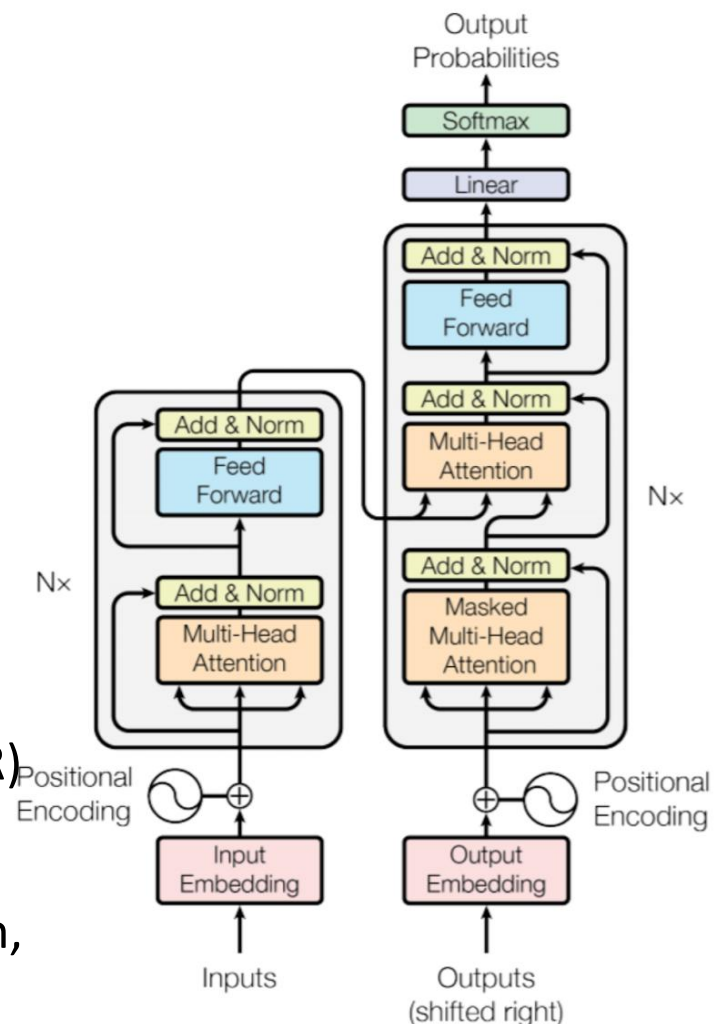
# Transformers – Architecture and Principles

## Transformer Components:

1. **Embeddings:** Convert tokens to vector representations
2. **Positional Encoding:** Adds position information
3. **Multi-Head Attention:** Processes relationships from multiple perspectives
4. **Feed-Forward Networks:** Process each position independently
5. **Layer Normalization:** Stabilizes training
6. **Residual Connections:** Helps with gradient flow

## Architecture Variations:

- **Encoder-only** (BERT): Good for understanding (classification, NER)
- **Decoder-only** (GPT): Good for generation
- **Encoder-decoder** (T5): Good for transformation tasks (translation, summarization)



# A Summary of how the Transformer Works

**Input Sentence:** "The cat chased the mouse."

## Input Encoding:

- Break down the sentence into tokens (words).
- Convert each token into a numerical representation called an embedding.
- Add positional encodings to the embeddings to provide information about the position of each token.

## Encoder Processing:

- The encoder takes the input embeddings with positional encodings.
- Pass the input through multiple layers of multi-head attention and feed-forward networks.
- Each encoder layer processes the input, allowing the model to learn complex representations of the sentence.
- The attention mechanism in the encoder learns relationships between tokens (e.g., "cat" is related to "chased" and "mouse").

## Decoder Processing:

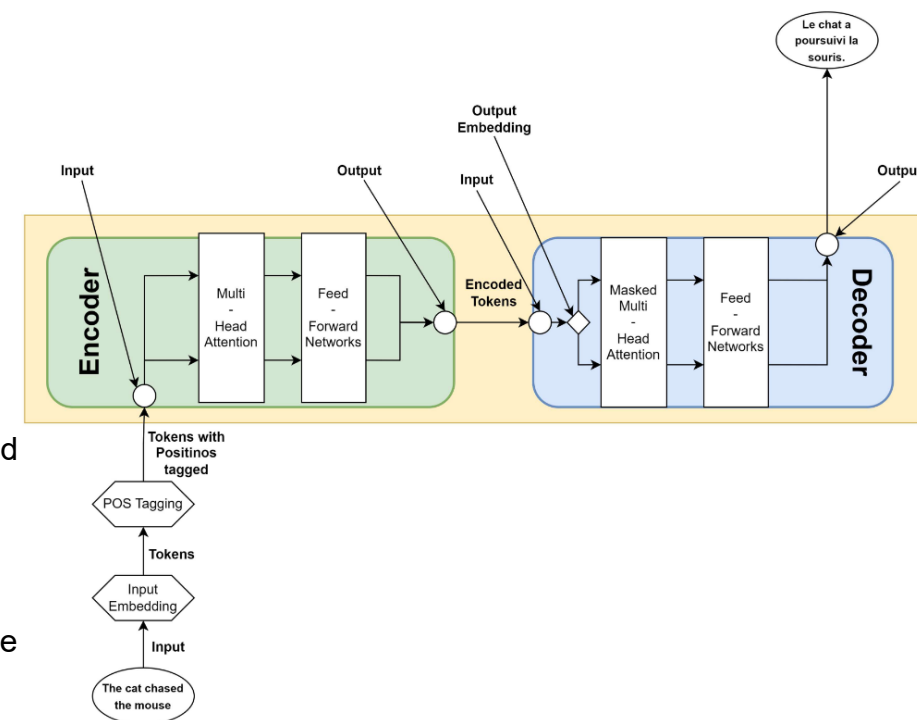
- The decoder takes the encoder's output.
- Use masked multi-head attention to generate the output one token at a time.
- Attend to the encoder's output to incorporate context from the input sentence while generating the translation.
- The attention mechanism in the decoder focuses on relevant parts of the input (e.g., the representation of "cat" when generating "Le chat").

## Output Generation:

- The decoder generates the output sequence token by token.
- For the example, it generates: "Le", "chat", "a", "poursuivi", "la", and "souris".
- The complete French translation is: "Le chat a poursuivi la souris."

## Key Advantages:

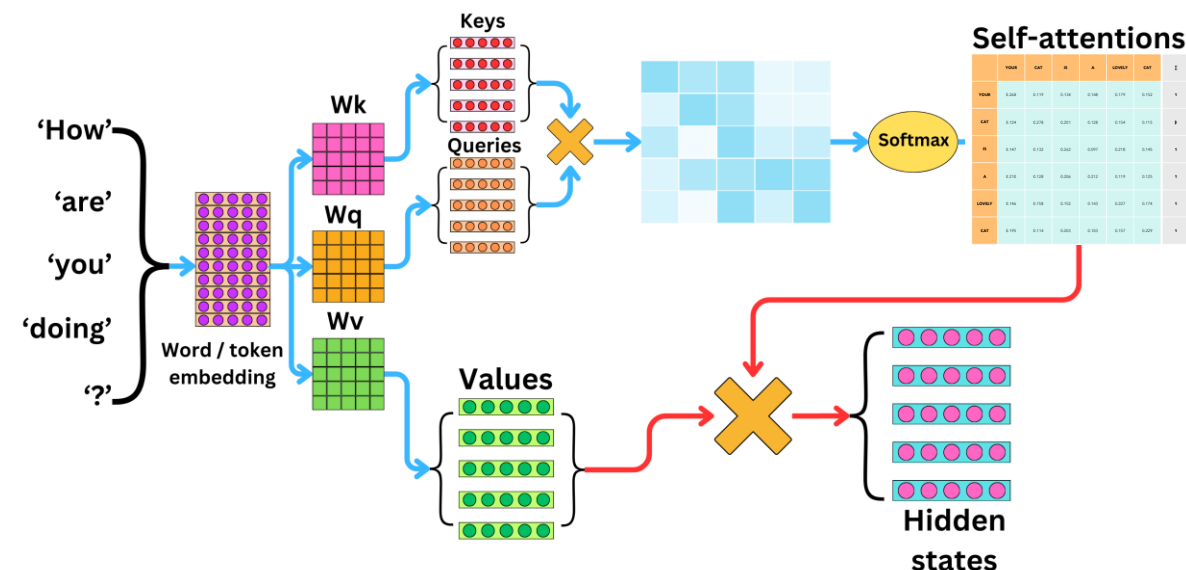
- Process the entire input sequence simultaneously.
- Use attention mechanisms to capture relationships between tokens.
- Efficiently translates sentences, even with long-range dependencies.



# The Attention Mechanism

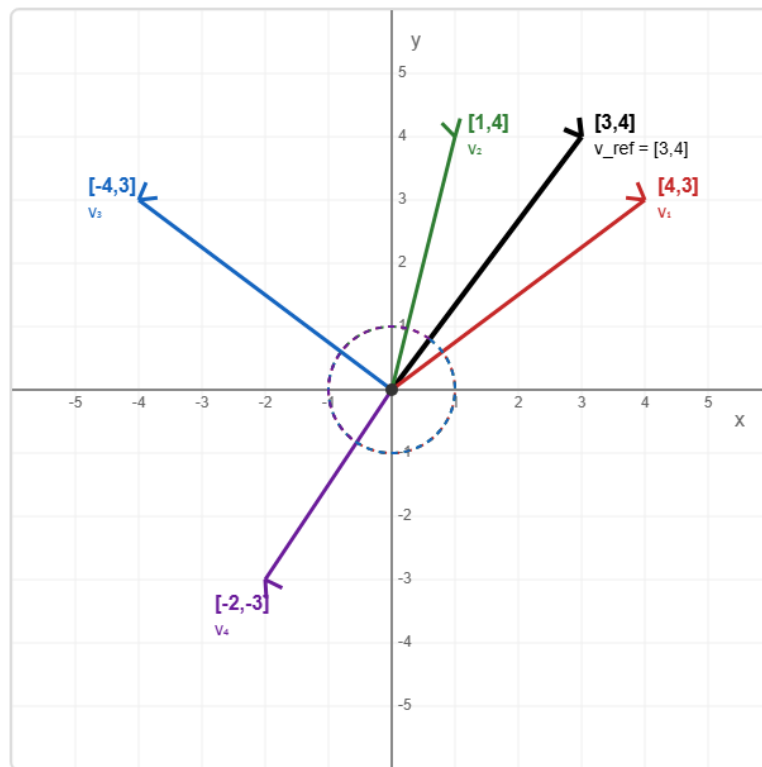
- Create Q, K, V matrices:** Each word embedding is multiplied by three learned weight matrices ( $W_Q$ ,  $W_K$ ,  $W_V$ ) to create Query, Key, and Value representations:
  - $Q = X \cdot W_Q$
  - $K = X \cdot W_K$
  - $V = X \cdot W_V$
- Compute attention scores:** Each query vector is dot-producted with all key vectors to get raw attention scores:
  - Scores =  $Q \cdot K^T$
- Scale the scores:** Divide by  $\sqrt{d_k}$  (where  $d_k$  is the dimension of the key vectors) to prevent the values from getting too large:
  - Scaled scores =  $(Q \cdot K^T) / \sqrt{d_k}$
- Apply softmax:** Convert the scaled scores to probabilities:
  - Attention weights =  $\text{softmax}(\text{Scaled scores})$
- Compute weighted values:** Multiply the attention weights by the value vectors:
  - Output = Attention weights  $\cdot V$

So the formula is:  $\text{Attention}(Q, K, V) = \text{softmax}((Q \cdot K^T) / \sqrt{d_k}) \cdot V$



# Vector Dot Product as Similarity Measure

$$\mathbf{v}_1 \cdot \mathbf{v}_2 = x_1x_2 + y_1y_2$$



## 1. Very Similar (High Positive)

Red Vector  $\mathbf{v}_1 = [4, 3]$

$$\mathbf{v}_1 \cdot \mathbf{v}_{\text{ref}} = (4 \times 3) + (3 \times 4) = 24$$

Almost same direction as reference

## 2. Similar (Positive)

Green Vector  $\mathbf{v}_2 = [1, 4]$

$$\mathbf{v}_2 \cdot \mathbf{v}_{\text{ref}} = (1 \times 3) + (4 \times 4) = 19$$

Generally same direction

## 3. Neutral (Zero)

Blue Vector  $\mathbf{v}_3 = [-4, 3]$

$$\mathbf{v}_3 \cdot \mathbf{v}_{\text{ref}} = (-4 \times 3) + (3 \times 4) = 0$$

Perpendicular (90° angle)

## 4. Dissimilar (Negative)

Purple Vector  $\mathbf{v}_4 = [-2, -3]$

$$\mathbf{v}_4 \cdot \mathbf{v}_{\text{ref}} = (-2 \times 3) + (-3 \times 4) = -18$$

Opposite direction



# Key Innovation: Self-Attention Mechanism

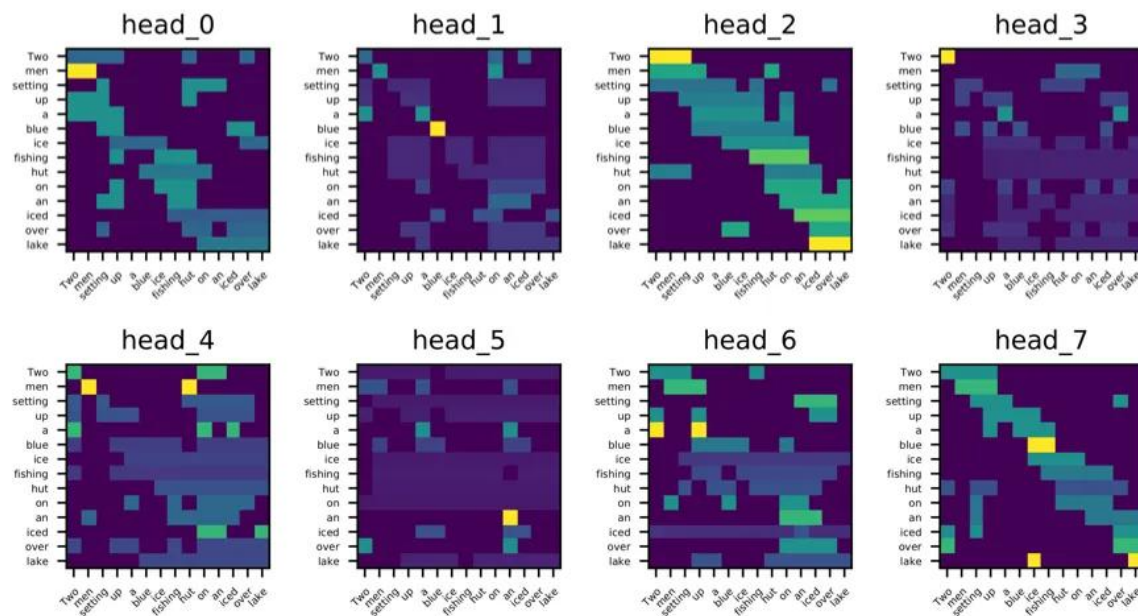
- Self-attention allows each word to compute its embedding by gathering information from all other words in the sequence.
- The "attention weights" determine how much each word should focus on other words.
- Words that are semantically related tend to have higher attention scores between them.
- This mechanism helps capture long-range dependencies and relationships regardless of word distance.
- Multiple attention heads in parallel (Multi-head Attention) allow the model to focus on different aspects of relationships.

	YOUR	CAT	IS	A	LOVELY	CAT	$\Sigma$
YOUR	0.268	0.119	0.134	0.148	0.179	0.152	1
CAT	0.124	0.278	0.201	0.128	0.154	0.115	1
IS	0.147	0.132	0.262	0.097	0.218	0.145	1
A	0.210	0.128	0.206	0.212	0.119	0.125	1
LOVELY	0.146	0.158	0.152	0.143	0.227	0.174	1
CAT	0.195	0.114	0.203	0.103	0.157	0.229	1



# Multiple Attention Heads

- Multiple attention heads in parallel (Multi-head Attention) allow the model to focus on different aspects of relationships.
- For example, one head might learn which words are related by grammar, while another might focus on semantic meaning.
- This allows the model to capture a richer and more comprehensive understanding of the input.

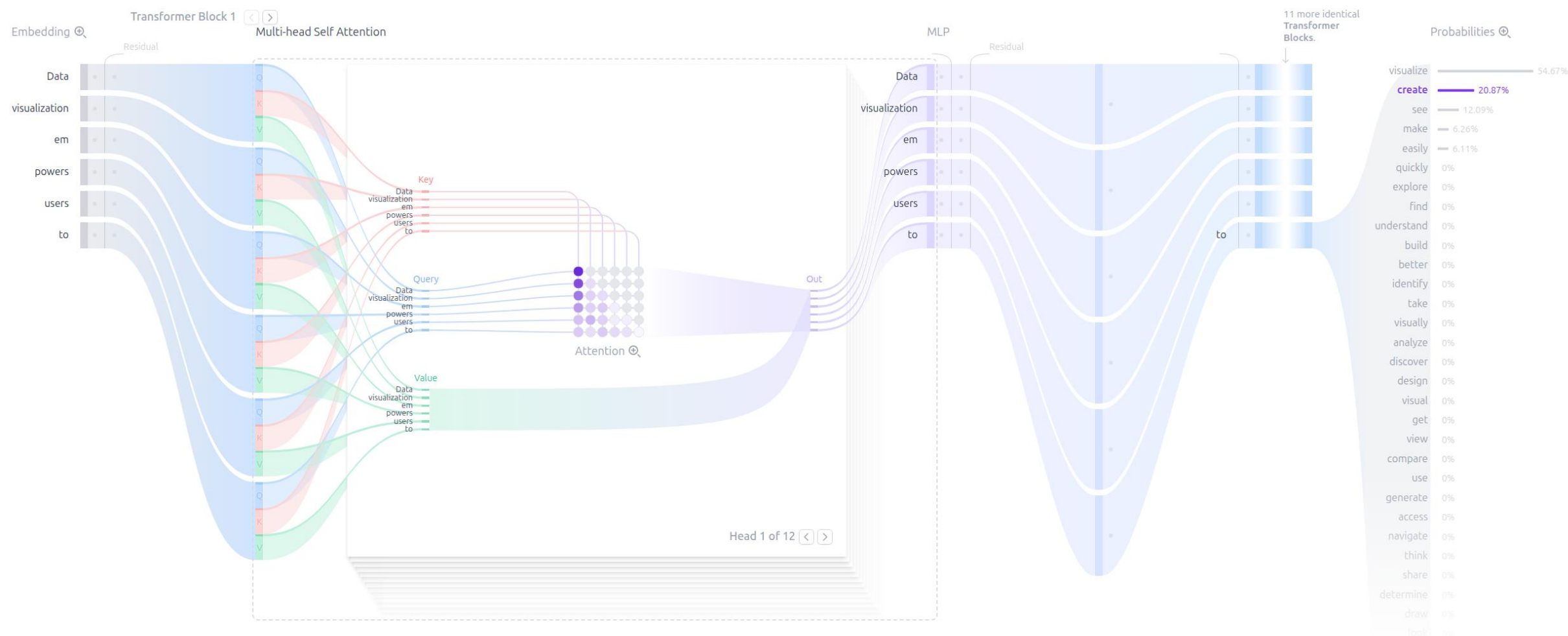




Component	Parameter	Formula / Size	Total Parameters
<b>Input</b>	Token embedding	$\text{Vocab\_Size} \times d\_model = 37000 \times 512$	$\approx 18.94\text{M}$
	Positional encoding (fixed)	$n \times d\_model$	Not learned (original paper used fixed)
<b>Attention (per layer)</b>	Q/K/V weights per head	$3 \times d\_model \times d\_k = 3 \times 512 \times 64$	98,304
	Output projection	$d\_model \times d\_model = 512 \times 512$	262,144
	<b>Total per Multi-Head block</b>	–	$\approx 360\text{K}$
<b>Feed-Forward (per layer)</b>	Linear 1: $512 \times 2048$	–	1,048,576
	Linear 2: $2048 \times 512$	–	1,048,576
	<b>Total FFN per layer</b>	–	$\approx 2.10\text{M}$
<b>LayerNorm</b>	$2 \times \gamma, \beta$ per layer	$2 \times d\_model = 2 \times 512$	1,024
<b>Encoder Block Total</b>	–	Attention + FFN + LayerNorm	$\approx 2.46\text{M}$
<b>Encoder Total (6 layers)</b>	–	$6 \times 2.46\text{M}$	$\approx 14.76\text{M}$
<b>Decoder Total (6 layers)</b>	Similar structure + cross attention	$\approx 2.6\text{M}$ per layer	$\approx 15.6\text{M}$
<b>Output Layer</b>	$d\_model \times \text{Vocab\_Size} = 512 \times 37000$	tied/shared with embedding	$\approx 18.94\text{M}$
<b>Total Model Parameters</b>	–	Encoder + Decoder + Embedding + Output	$\approx 65\text{M}$

# Transformer Explainer

<https://poloclub.github.io/transformer-explainer/>



# Context Aware Embeddings

# Contextual Word Embeddings (BERT and GPT)

## Key Innovation

Unlike static embeddings (Word2Vec, GloVe), contextual models generate **different vectors for the same word** depending on its context.

## Approach

- Uses deep, pre-trained neural networks (often transformer-based)
- Embeddings are derived from entire sentences, capturing syntax and semantics dynamically

## Examples

- **BERT - Bidirectional Encoder Representations from Transformers (2018):** Transformer-based neural networks trained with masked language modeling and next sentence prediction
- **GPT - Generative Pre-training Transformer (2018):** Transformer-based unidirectional language model focused on generation.

## Characteristics

- Embeddings are **context-sensitive** (e.g., “bank” in “river bank” vs. “savings bank”)
- Each word is embedded based on its role in the sentence.
- Embeddings vary for the same word depending on its position and meaning.
- Significantly improve performance on downstream NLP tasks.

# BERT: Bidirectional Encoder Representations from Transformers

- Developed by Google in 2018
- A **pre-trained language model** based on the **Transformer encoder**
- Reads text **bidirectionally**, enabling deep contextual understanding

## Key Ideas

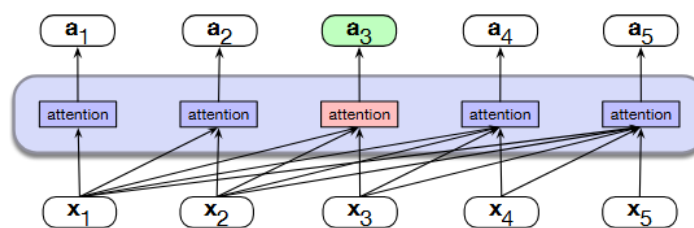
- Uses only the **encoder** stack of the Transformer
- Pre-trained on large text corpora, then fine-tuned on specific tasks

## Pretraining Objectives

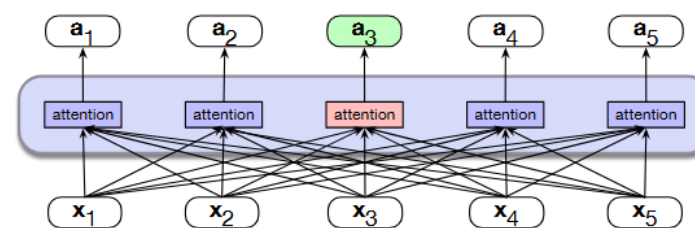
- **Masked Language Modeling (MLM)**: Predict randomly masked words in a sentence
- **Next Sentence Prediction (NSP)**: Predict if one sentence follows another

## Applications

- Sentiment Analysis
- Question Answering
- Named Entity Recognition
- Text Classification
- Semantic Search



a) A causal self-attention layer



b) A bidirectional self-attention layer

# More About BERT

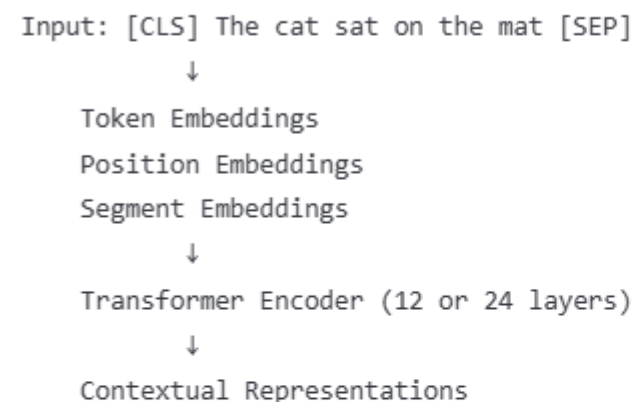
## Key Advantages

- Contextual embeddings: Word meanings change based on context
- Captures long-range dependencies
- Pre-trained on massive datasets → Transfer learning
- State-of-the-art performance on 11 NLP tasks when released

## How BERT Works:

- **Multiple stacked encoder layers** with self-attention mechanisms
- **No decoders** - purely focused on understanding input
- **Captures relationships** between words and their context
- **Powerful context learning** - understands word relationships in sentences
- **Meaningful representations** - transforms text into useful numbers

## Architecture Overview



## Key Components

- **[CLS]**: Classification token (sentence-level tasks)
- **[SEP]**: Separator token (between sentences)
- **Multi-head attention**: Allows model to focus on different positions
- **Feed-forward networks**: Process attention outputs

# BERT's Training Data

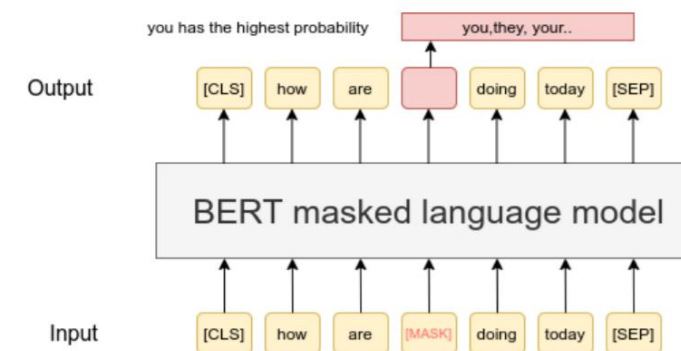
## Pre-training Foundation

- **Training corpus:** 3+ billion words
  - English Wikipedia
  - ~10,000 unpublished books
- **Clean, large-scale dataset** for comprehensive language learning
- **Pre-training approach** to learn language patterns and context

# Training Objective 1 - Masked Language Modeling

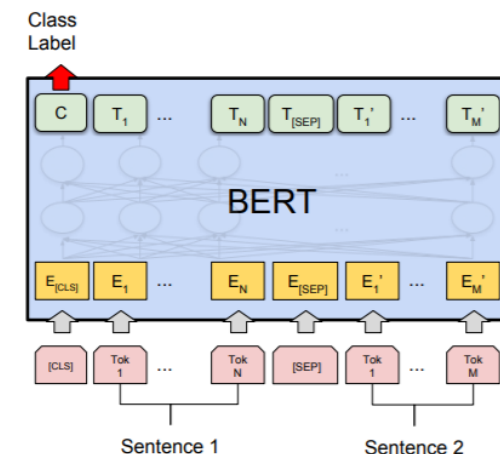
## Learning Through "Fill in the Blanks"

- **Random word masking:** Some words are hidden during training
- **Context-based prediction:** BERT predicts masked words using surrounding context
- **Example:** "I love eating \_\_\_\_\_ in my fruit salad" → "lemons"
- **Key benefit:** Learns word relationships and contextual understanding
- **Powerful concept:** Used in many other AI applications



## Training Objective 2 - Next Sentence Prediction

- **Title: Understanding Sentence Relationships**
- **Sentence pair analysis:** Given two sentences, determine if B follows A
- **Logical connection assessment:** Analyzes context and content
- **Example:**
  - A: "The cat climbed the tree"
  - B: "It was trying to catch a bird" ✓ (follows logically)
  - C: "The weather is nice today" ✗ (unrelated)
- **Note:** Later removed in newer models (MLM proved sufficient)



(a) Sentence Pair Classification Tasks:  
MNLI, QQP, QNLI, STS-B, MRPC,  
RTE, SWAG

# BERT's Ideal Use Cases

## Where BERT Excels

- **Text Classification:** Sentiment analysis, topic classification, spam detection
- **Named Entity Recognition:** Identifying people, organizations, locations, dates
- **Extractive Question Answering:** Finding answers within provided context
- **Semantic Similarity:** Measuring similarity between sentences/paragraphs
- **Key advantage:** Perfect for understanding tasks, not generation

# Extractive Question Answering Example

## How BERT Answers Questions:

- **Process:** Analyzes both context and question
- **Method:** Identifies start and end tokens of the answer
- **Example:**
  - Context: "Mount Everest is the highest mountain..."
  - Question: "What is the highest mountain?"
  - Answer: "Mount Everest" (extracted, not generated)
- **Important:** BERT extracts existing text, doesn't generate new content

Component	BERT-Base (L=12, H=768, A=12)	BERT-Large (L=24, H=1024, A=16)
<b>Embedding Layer</b> - Token + Positional + Segment	$30522 \times 768 + 512 \times 768 + 2 \times 768 \approx 23.8\text{M}$	$30522 \times 1024 + 512 \times 1024 + 2 \times 1024 \approx 31.4\text{M}$
<b>Self-Attention</b> - Q/K/V + Output per layer	$4 \times H^2 = 4 \times 768^2 = 2.36\text{M}$	$4 \times 1024^2 = 4.19\text{M}$
Total across all layers	$12 \times 2.36\text{M} = 28.3\text{M}$	$24 \times 4.19\text{M} = 100.6\text{M}$
<b>Feedforward</b> - Two linear layers per layer	$768 \times 3072 + 3072 \times 768 = 4.71\text{M}$	$1024 \times 4096 \times 2 = 8.39\text{M}$
Total across all layers	$12 \times 4.71\text{M} = 56.5\text{M}$	$24 \times 8.39\text{M} = 201.4\text{M}$
<b>LayerNorms</b> - Two per layer	$2 \times 768 = 1.5\text{K} \times 12 = 18\text{K}$	$2 \times 1024 \times 24 = 49\text{K}$
<b>Pooler</b> - Final CLS output to 768 or 1024	$768 \times 768 = 0.59\text{M}$	$1024 \times 1024 = 1.05\text{M}$
<b>Total Parameters</b>	<b><math>\approx 110\text{M}</math></b>	<b><math>\approx 340\text{M}</math></b>

- **L: number of layers (Transformer blocks)**
- **H: hidden size**
- **A: number of attention heads (H / A = size per head)**
- **Vocabulary size: 30,522**
- **FFN hidden size: 4×H (3072 for base, 4096 for large)**

# BERT Variants

## Common BERT Models

Model	Parameters	Layers	Hidden Size	Attention Heads
BERT-Base	110M	12	768	12
BERT-Large	340M	24	1024	16
DistilBERT	66M	6	768	12
RoBERTa	355M	24	1024	16
ALBERT	12M-235M	12-24	768-4096	12-64

## Specialized Variants

- **BioBERT**: Biomedical text
- **SciBERT**: Scientific text
- **FinBERT**: Financial text
- **ClinicalBERT**: Clinical notes

## Example: Print out BERT Embeddings

```
from transformers import BertTokenizer, BertModel
import torch

# Load pretrained BERT
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
model = BertModel.from_pretrained('bert-base-uncased')

# Sentence
sentence = "He went to the bank to deposit money."

# Tokenize
inputs = tokenizer(sentence, return_tensors='pt')

# Get outputs
with torch.no_grad(): # No training, just inference
    outputs = model(**inputs)

# Get the hidden states (embeddings)
embeddings = outputs.last_hidden_state # Shape: (batch_size, sequence_length,

# (hidden_size)
print(embeddings.shape) # Example output: torch.Size([1, 11, 768])
```

# GPT – Overview and Architecture

## What is Generative Pre-trained Transformer (GPT)?

- A family of **Transformer-based language models** developed by OpenAI
- Uses only the **decoder stack** of the original Transformer architecture
- Trained with **causal (autoregressive) language modeling** to predict the next token
- Focuses on generating human-like text (unlike BERT's understanding focus)

## Training Objective

- Predict the next token in a sequence

## GPT Variants

- **GPT-1 (2018, 120M Parameters)**: Introduced the pretrain-then-finetune paradigm
- **GPT-2 (2019, 1.5B Parameters)**: Scaled up model size, trained on web-scale data, Last publicly available weights
- **GPT-3 (2020, 175B Parameters)**: Enabled in-context learning, 100x boost, ~800GB file size
- **GPT-4 (2023, ~1T parameters (estimated))**: Multimodal, stronger reasoning and generalization

## Applications (Wide range of NLP tasks through text generation)

- Text generation (e.g., chat, storytelling, code)
- Summarization
- Translation
- Question answering
- Semantic search and reasoning tasks

# GPT's Architecture Deep Dive

## How GPT Works

- **Multiple stacked decoder layers** with self-attention mechanisms
- **No encoders** - purely focused on text generation
- **Unidirectional processing** - considers only left-side context
- **Next word prediction** based on preceding words
- **High-quality text generation** from learned context patterns
- **Contextual understanding** combined with text creation capability

# GPT Training Data

## Pre-training Foundation

- **Massive, diverse datasets** including:
  - Web pages
  - Books and articles
  - Billions of words total
- **Different datasets** for each GPT version
- **Large-scale exposure** to language patterns and context
- **Quality varies** but emphasis on diversity and scale

# Causal Language Modeling

## GPT's Single Training Objective

- **Core task:** Predict the next word in a sequence
- **Method:** Uses context from preceding words only
- **Example:** "I love drinking fresh \_\_\_\_" → "lemonade"
- **Key benefits:**
  - Learns word relationships and context
  - Develops language patterns
  - Enables coherent text generation
- **Simplicity:** No masked language modeling or next sentence prediction

# GPT's Ideal Use Cases

## Where GPT Excels

- **Text Generation:** Story creation, creative writing, conversations
- **Instruction Following:** Responding to prompts and commands
- **Translation:** Converting text between languages
- **Summarization:** Creating concise summaries of longer texts
- **Code Generation:** Programming assistance and code completion
- **Versatile Applications:** Any task involving text output



Component	Parameter	Formula / Size	Total Parameters (Approx.)
Embedding Layer	Token Embeddings	$\text{Vocab} \times d_{\text{model}} = 50\text{K} \times 12288$	614.4M
	Positional Embeddings	$n_{\text{ctx}} \times d_{\text{model}} = 2048 \times 12288$	25.2M
Self-Attention (per layer)	Q, K, V, Output	$4 \times d_{\text{model}} \times d_{\text{model}} = 4 \times 12288^2$	604.6M per layer
Feedforward (per layer)	2 layers (gelu)	$d_{\text{model}} \times d_{\text{ff}} + d_{\text{ff}} \times d_{\text{model}}$	1.2B per layer
LayerNorms (per layer)	Two per layer	$2 \times d_{\text{model}}$	24.6K per layer
Total per layer	–	Self-attn + FFN + norms	$\approx 1.8\text{B}$ per layer
Transformer Block Total	$96 \times 1.8\text{B}$	–	$\approx 172.8\text{B}$
Final LayerNorm	–	$d_{\text{model}}$	12.3K
Output Layer (tied)	Shared with token embedding	–	– (tied with input embedding)
Total Parameters	–	Sum of above	$\approx 175\text{B}$

- $d_{\text{model}} = 12288$
- $n_{\text{layers}} = 96$
- $n_{\text{heads}} = 96 \rightarrow$  each head has  $d_{\text{k}} = d_{\text{v}} = 128$
- $d_{\text{ff}} = 4 \times d_{\text{model}} = 49152$
- Vocabulary size  $\approx 50,000$
- Sequence length ( $n_{\text{ctx}}$ ) = 2048

# GPT vs BERT Architecture

Aspect	GPT	BERT
Architecture	Decoder-only	Encoder-only
Text Processing	Unidirectional (left-to-right)	Bidirectional
Primary Goal	Text generation	Text understanding
Context	Previous words only	All surrounding words
Use Cases	Generation tasks	Classification/extraction

## What is Hugging Face? 🤔

- **A company and a community platform** focused on democratizing Artificial Intelligence, especially Natural Language Processing (NLP) and Machine Learning (ML).
- Often called the "**GitHub for Machine Learning.**"
- **Mission:** To make state-of-the-art ML models, datasets, and tools accessible to everyone.
- Started in 2016, initially with a chatbot app, then pivoted to open-source ML.

### What does hugging face provide?

1. **Accessibility:** Provides easy access to thousands of pre-trained LLMs.
2. **Standardization:** Offers standardized tools and interfaces for working with different models.
3. **Collaboration:** Fosters a vibrant community for sharing models, datasets, and knowledge.
4. **Innovation:** Accelerates research and development in the LLM field.
5. **Ease of Use:** Simplifies complex ML workflows, from data preparation to model deployment.

# Core Components of the Hugging Face Ecosystem

- **Hugging Face Hub:**
  - The central place to find, share, and collaborate on models, datasets, and ML applications (Spaces).
  - Over 1.7 million models, 75,000+ datasets!
- **Transformers Library:**
  - Python library providing thousands of pre-trained models for NLP, Computer Vision, Audio, and more.
  - Supports PyTorch, TensorFlow, and JAX.
  - Makes downloading, training, and using state-of-the-art models incredibly simple.
- **Datasets Library:**
  - Efficiently load and process large datasets.
  - Optimized for speed and memory, built on Apache Arrow.
  - Access to a vast collection of public datasets.
- **Tokenizers Library:**
  - Provides high-performance tokenizers crucial for preparing text data for LLMs.
  - Offers various tokenization algorithms and pre-trained tokenizers.

# The Model Hub

**Over 1,700,000+ Models Available**

**Popular Model Categories:**

- **Text Generation:** GPT, LLaMA, Mistral, CodeLlama
- **Text Classification:** BERT, RoBERTa, DeBERTa
- **Question Answering:** BERT-based models
- **Translation:** T5, mT5, NLLB
- **Code Generation:** CodeT5, StarCoder
- **Multimodal:** CLIP, BLIP, LLaVA

# Getting Started with Hugging Face

- Explore the Hub: [huggingface.co](https://huggingface.co)
- Browse models, datasets, and Spaces.
- Install Libraries:

```
pip install transformers datasets tokenizers  
accelerate gradio
```

- Try a Pipeline:

## # Example: Sentiment Analysis

```
from transformers import pipeline  
  
classifier = pipeline("sentiment-analysis")  
  
result = classifier("Hugging Face is awesome!")  
  
print(result)
```

## # Example: Text Generation

```
generator = pipeline("text-generation")  
  
output = generator("In a world of large language models,",  
                    max_length=50)  
  
print(output)
```

### Under the Hood

1. Automatic model selection
2. Tokenization handled
3. Inference optimization
4. Result formatting
5. Device management

### Traditional Approach

1. Load tokenizer
  2. Preprocess text
  3. Load model
  4. Run inference
  5. Post-process results
- ... 50+ lines of code

# Other Hugging face Pipelines

The Hugging Face `transformers` library supports a wide range of **pipelines**, each designed for a specific **natural language processing (NLP)** or **vision task** — so you can use powerful models without deep setup.

Pipeline Name	Task Description
"sentiment-analysis"	Classify sentiment (positive/negative)
"text-classification"	General text classification (multi-label or multi-class)
"zero-shot-classification"	Classify into labels <b>without training</b> on them
"text-generation"	Generate text (e.g., GPT models)
"text2text-generation"	Text-to-text tasks (e.g., summarization, translation)
"translation"	Translate between languages
"summarization"	Generate a summary of input text
"question-answering"	Extract answer from context
"fill-mask"	Predict missing word in a sentence (BERT-style)
"ner" (Named Entity Recognition)	Extract entities (like names, places, etc.)
"conversational"	Chatbot-style conversation
"sentence-similarity"	Measure similarity between two sentences
"token-classification"	Classify each token (used for NER, POS tagging, etc.)
"feature-extraction"	Extract embeddings/features from a model
"table-question-answering"	QA over structured data (tables)

## ► Sentiment Analysis

```
python
pipeline("sentiment-analysis")("I love this!")
```

## ► Summarization

```
python
pipeline("summarization")("Long article text goes here...")
```

## ► Translation

```
python
pipeline("translation_en_to_fr")("This is amazing.")
```

## ► Question Answering

```
python
qa = pipeline("question-answering")
qa({
    "question": "Where do pandas live?",
    "context": "Pandas are native to China and prefer bamboo forests."
})
```

To list all available pipelines in code:

```
python
from transformers.pipelines import SUPPORTED_TASKS
print(SUPPORTED_TASKS.keys())
```

# Training LLMs

# What is LLM Training?

- LLM Training is the process of teaching a neural network to understand, generate, and manipulate human language.
- It involves feeding the model vast amounts of text data.
- The model learns patterns, grammar, context, and even some level of "knowledge" from this data.
- The goal is to adjust the model's internal parameters (weights and biases) to perform specific language tasks effectively.
- Modern models have BILLIONS of parameters (numbers) to learn during the training process.

# Essential Components of LLM Training

1. **Dataset:** Large corpus of text data (e.g., books, articles, websites). Quality, quantity, and diversity are crucial.
2. **Model Architecture:** The neural network structure, predominantly the Transformer architecture (with self-attention mechanisms).
3. **Loss Function:** A function that measures the difference between the model's predictions and the actual target values (e.g., cross-entropy for next-word prediction).
4. **Optimizer:** An algorithm that updates the model's parameters to minimize the loss function (e.g., Adam, SGD).

# The General Training Loop

1. **Data Preparation:** Collecting, cleaning, and tokenizing the text data into a format the model can understand.
2. **Model Initialization:** Setting initial random values for the model's parameters.
3. **Forward Pass:** Feeding input data through the model to get predictions.
4. **Loss Calculation:** Comparing predictions to the actual data to quantify error using the loss function.
5. **Backward Pass (Backpropagation):** Calculating gradients, which indicate how each parameter contributed to the error.
6. **Parameter Update:** Adjusting model parameters using the optimizer in the direction that reduces the loss.
7. **Iteration:** Repeating steps 3-6 for many batches of data over multiple epochs (passes through the entire dataset).

# Main Training Phases

1. LLM Pretraining.
2. LLM Fine Tuning

# Pre-training: Building the Foundation

- The initial, **most resource-intensive** training phase.
- Models are trained on massive, diverse datasets (e.g., Common Crawl, Wikipedia, books).
- The objective of this step is to learn general language understanding, grammar, common sense reasoning, and factual knowledge.
- Usually self-supervised (e.g., predicting masked words, next sentence prediction).
- Results in a **base model** with broad capabilities.
- Examples: GPT-3, BERT, Llama pre-training.

## Fine-tuning: Specializing the Model

- Takes a pre-trained base model and **further trains it on a smaller, task-specific dataset**.
- The objective of this step is to **adapt the general knowledge** of the pre-trained model to perform well on a particular downstream task (e.g., medical question answering, legal document summarization).
- It **requires significantly less data and computation** than pre-training.
- Can also be used for instruction tuning (following prompts) or aligning with human preferences (RLHF).

# Computational Demands & Challenges

- **Hardware:** Requires powerful GPUs (Graphics Processing Units) or TPUs (Tensor Processing Units) for parallel computation.
- **Distributed Training:** Often necessary to train large models across multiple GPUs or machines, adding complexity.
- **Time:** Pre-training can take weeks or months, even with significant computational resources.
- **Cost:** Significant expenses for hardware, cloud computing, and energy consumption.
- **Memory:** Model parameters and activations require substantial memory. Techniques like mixed-precision training help.

# LLM Training Details - More Recent Models & Considerations

Model (Version/Size)	Est. Data Size (Tokens)	Context Size (Max Tokens)	Est. GPUs / Compute	Est. Training Time	Est. Electricity / Carbon Footprint	Est. Training Cost
GPT-3 (Base models like Davinci)	~500 Billion - 1 Trillion (incl. C4, Wikipedia, Books, etc.)	2,048 (later models up to 4,096)	~10,000 V100 GPUs (for original run) / ~3.6M A100- equivalent hours (PaLM 540B reference)	~34 days (using 1,024 A100s, research estimate) - Several months (actual, unconfirmed)	~1,287 MWh (training) / ~552 tons CO <sub>2</sub> eq (Patterson et al.)	\$4.6M - \$12M+ (compute, various estimates)
Llama 2 (All sizes)	2 Trillion	4,096	Reported 6,000 GPU- months (A100-80GB equivalent for the family) / 1.7M+ GPU hours for 70B	Jan 2023 - July 2023 (overall project, specific model run time shorter within this)	3.3M kWh (entire project) / 539 tons CO <sub>2</sub> eq (training, 100% offset by Meta)	Significant (part of Meta's AI investment)
BLOOM (176B)	366 Billion (1.6 TB)	2,048	384 A100 GPUs	~3.5 - 4 months	~433 MWh (training) / ~25- 55 tons CO <sub>2</sub> eq (trained in France, low- carbon energy)	~\$2M - \$5M (compute, public estimates)

# LLM Training Details - More Recent Models & Considerations

Model (Version/Size)	Est. Data Size (Tokens)	Context Size (Max Tokens)	Est. GPUs / Compute	Est. Training Time	Est. Electricity / Carbon Footprint	Est. Training Cost
GPT-4	Not officially disclosed (speculated >> GPT-3, likely multi-trillion)	8,192 (GPT-4-8k) & 32,768 (GPT-4-32k); GPT-4 Turbo: 128,000	Not officially disclosed (speculated tens of thousands of A100s/H100s)	~5-6 months (speculative estimates)	Not disclosed (Expected to be significantly higher than GPT-3; estimates range from 20,000-78,000 MWh & thousands of tons CO <sub>2</sub> eq for comparable efforts)	Est. >\$60M - \$100M+ (compute, speculative)
Llama 3 (Instruct models)	>15 Trillion (for the Llama 3 family)	8,192 (some reports suggest up to 128k for future/experimental versions)	Significant clusters of H100s (e.g., Meta mentioned two 24k H100 clusters)	~3 days (8B), ~17 days (70B), ~97 days (est. for 400B+ on 16k H100s)	Llama 3.1 405B est. ~11 GWh. Carbon footprint not yet fully disclosed, but Meta aims for net-zero operations.	Very High (part of Meta's large AI infrastructure investment)
Gemini 1.0 (Pro/Ultra)	Not officially disclosed (multimodal, likely vast & diverse datasets)	32,768 (Gemini 1.0 Pro); Gemini 1.5 Pro: 1 Million (up to 10M experimental)	Trained on Google's TPU v4 and v5e pods (thousands to tens of thousands of TPUs)	Not publicly disclosed (likely months)	Not disclosed. Google emphasizes efficiency & use of renewable energy. Gemini 1.0 was reported to be more efficient than some predecessors.	Very High (part of Google DeepMind's core AI efforts)

# Fine-Tuning Large Language Models (LLMs)

# What is Fine-Tuning?

- **Pre-training Phase**
  - Trained on large corpus using Masked Language Modeling (MLM) and Next Sentence Prediction (NSP)
- **Fine-Tuning Phase**
  - Fine-tuning is the process of continuing the training of a pretrained LLM on a smaller, task-specific dataset.
  - The objective is to specialize the model for a particular use case or domain.
- **Why Fine Tune LLMs?**
  1. Improve performance on specific tasks.
  2. Inject domain-specific knowledge (legal, medical, financial, etc.).
  3. Adapt to company-specific language or tone.
  4. Reduce inference cost by limiting model size and scope.
- **Use Cases of BERT Fine-Tuning**
  - Customer Support Chatbots (trained on company FAQs).
  - Legal Document Analysis.
  - Scientific Paper Summarization.
  - Code Assistants for specific frameworks.
  - Sentiment classification for product reviews.

# Typical LLM Fine Tuning Workflow

## 1. Choose a Pre-trained Model

Pick a base model from Hugging Face (e.g., distilbert-base-uncased for classification or gpt2 for text generation).

## 2. Prepare Your Dataset

Format your data appropriately. Common formats:

- For classification: CSV with text and label columns.
- For generation: Text file or JSON with prompt and completion.

## 3. Tokenize the Data

Convert text into tokens the model understands using a tokenizer.

## 4. Define Training Arguments

Set training parameters like learning rate, batch size, and number of epochs.

## 5. Train the Model

Use a trainer to fine-tune the model on your dataset.

## 6. Evaluate & Save

Check performance and save the model for future use.

## 7. Share (Optional):

- Push your fine-tuned model back to the Model Hub.
- Create a demo in Hugging Face Spaces.